CONIKS: Preserving Secure Communication with Untrusted Identity Providers

Marcela S. Melara

Master's Thesis

Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Master of Science in Engineering

Recommended for Acceptance
by the Department of Computer Science
Princeton University

Adviser: Edward W. Felten

June 2014

© Copyright by Marcela S. Melara, 2014.

All rights reserved.

Abstract

As a foundation for securing Internet communication, today's public-key infrastructures attempt to certify that the "correct" principal owns a name and possesses a corresponding public key. Yet these assertions are not machine-verifiable, and their correctness can be and has been broken by error, malice, or external (and sometimes legal) coercion. Rather than focus on the "correctness" of name ownership, we argue that ensuring the *continuity* of that ownership, as defined by the binding of a name to a public key, is necessary and often sufficient for secure communication.

To this end, we have designed CONIKS, a key management system that ensures online identity providers maintain and respond with valid bindings from names to cryptographic keys, even without users managing (or seeing) those keys. If providers disseminate invalid bindings or equivocate by presenting different bindings to different users, parties quickly detect this misbehavior and can publish irrefutable proof of this equivocation. The cryptographic mechanisms employed by CONIKS are highly efficient, supporting an Internet environment of millions of providers, any of which may have users in the billions. ConiksChat, our secure chat service prototype integrates CONIKS into an existing chat application and demonstrates that one can achieve high usability while preserving the security and privacy of online communications.

Acknowledgements

First of all, I would like to thank Ed Felten and Mike Freedman for co-advising me on this project. They gave me the support and advice I needed to gain an intuition for computer security and systems research, and I could have not gotten ready for a PhD without their education and care. I hope these next several years at Princeton will be as enlightening and fruitful as the my first two years.

I would also like to thank Aaron Blankstein for being a great collaborator and mentor on this project. All of his insight on the problems we worked to solve, but also in general on the ins and outs of conducting research were extremely helpful. I really appreciate his willingness to hear me out whenever I had questions or concerns. I would also like to thank Josh Kroll for his guidance on the prototype implementation of CONIKS. His continuing support of this project was very helpful in getting the project to where it is today.

Of course, I also have to thank all of my Hobart and William Smith Professors for realizing that I would want to take the research route before I did. I am grateful for their unbounded encouragement and guidance towards where I am today. I remember all of our conversations about school and life very fondly.

Most importantly, I am grateful for all of my family and friends. My parents and sister are always there when I need a pep-talk, advice or just a place to vent. All of my CITP, Princeton, HWS and Erding friends have been invaluable to keeping me grounded and giving me reasons to laugh when things get most stressful. And I would like to thank Wade for his support, understanding and love. I am happy to have him in my life, and I owe my sanity to him. I look forward to facing the challenges of the next few years together.

Contents

Abstract . . . iii

Acknowledgements . . . iv

List of Figures . . . viii

1 Introduction **1**

1.1 A Different Approach to Identity: Continuity . . . 2

1.2 Continuity vs. Correctness . . . 4

2 Related Work **6**

2.1 Certificate Transparency . . . 6

2.2 Keybase . . . 8

2.3 OneName . . . 9

2.4 Certificate Validation Systems . . . 10

2.5 Identity Services and Authentication Systems . . . 11

2.6 Alternative Public-Key Infrastructures . . . 12

2.7 Untrusted Cloud Services . . . 13

3 CONIKS Overview **14**

3.1 Threat Model . . . 16

3.1.1 Identity Provider . . . 16

3.1.2 Clients . . . 17

3.1.3 Third-party entities . . . 17

3.2 CONIKS Properties 18
3.2.1 Whistleblowing 18
3.3 Enabling Continuity Checks 19
3.4 Applications 20
3.4.1 Secure Webmail 20
3.4.2 Secure Communications 21
3.4.3 Secure Credentials 21
3.4.4 Adopting CONIKS in practice 22

4 System Design **24**
4.1 Making Bindings Verifiable 26
4.1.1 Committing to a Merkle Search Tree 26
4.1.2 Comparable Commitments 27
4.1.3 Associating Commitments over Time 27
4.2 Continuity Checks 28
4.2.1 Checking for Commitment Non-Equivocation 28
4.2.2 Checking for Binding Validity 29
4.2.3 Performing Checks after Missing Epochs 30
4.3 Whistleblowing 31

5 Security Analysis **33**
5.1 Detecting Equivocation 33
5.2 Ensuring Validity 36
5.3 Limiting the Effects of Denied Service 37

6 Discussion **38**
6.1 Scaling CONIKS 38
6.1.1 Computing and Transmitting Commitments 38
6.1.2 Commitment Verification 39

6.1.3 Distributed Operation by Identity Providers. 39
6.2 Caching Key Information 40
6.3 Key Loss and Account Protection 40
6.4 Protocol Extensions 41
6.4.1 Concealing the Number of Users 41
6.4.2 Returning Updated Bindings between Epochs 41

7 Prototype Implementation 43
7.1 Background: XMPP, OTR and Tigase 43
7.1.1 XMPP 43
7.1.2 Off-the-Record Messaging 44
7.1.3 Tigase XMPP/Jabber Server 45
7.2 ConiksChat Implementation Details 45
7.2.1 ConiksChat Service 45
7.2.2 ConiksChat Client 47
7.2.3 CONIKS Protocol Buffers 49
7.2.4 Preliminary Evaluation and Discussion 49

8 Conclusion 51

Bibliography 53

List of Figures

3.1 User-provider interactions during secure communication. 15

4.1 User-provider interactions during Continuity Checks. 25

4.2 An authentication path in a namespace tree. 26

4.3 A hash chain of commitments. 27

4.4 Flow chart of the hash chain verification. 28

4.5 Flow chart of the witnessed commitments check. 28

4.6 Flow chart of the binding validity check. 29

5.1 Probability of two users detecting an equivocation. 35

Chapter 1

Introduction

Today, billions of users depend on online services to communicate with each other. Rather than running their own servers, users rely on these providers for their ease of use, global accessibility, and high availability. Yet recent revelations about system penetrations by outside attackers [26, 28], and mass online surveillance and data collection by government agencies [11, 20, 24] are a reminder that these centralized services are high-value targets for attack. While many services have taken a step towards securing their communication channels via encrypted HTTP and SMTP, users' data is typically stored unencrypted on the providers' servers, leaving the private messages vulnerable to security breaches [14, 41]. Even if providers confronted external attacks with heightened security mechanisms, these do not address *insider* attacks or coercion by governments or others. Users face a choice between the risks of centralized systems, and the challenges of running their own servers. A better approach would take advantage of centralized services while mitigating the risks of outsider and insider access to data.

In principle, users can mitigate these risks by adopting end-to-end encryption and authentication tools such as PGP [12, 68, 51]. These tools, however, require that users explicitly reason about cryptographic keys and encryption. For many users,

these concepts can be error-prone and difficult to adopt [18, 67]. In the decentralized "Web of Trust" model, a transitive chain of trust must be found between the two communicating entities. On the other hand, the traditional certificate authority (CA) model assumes that there exist universally trusted parties and that these parties can somehow verify the identity of users at low cost. Neither approach copes well with attacks that coerce a certifier into issuing a bogus certificate allowing the attacker to impersonate a targeted user and launch a man-in-the-middle attack.

1.1 A Different Approach to Identity: Continuity

Since certifiers are still capable of comprising the integrity and confidentiality of secure communication in the Web of Trust and the CA models, we propose to address this problem by changing what is required of a certifier. Rather than requiring that the binding of a key to a name be *correct* in the sense that the key belongs to the particular real-world person or entity designated by the name, a certifier should only attest that the binding is *continuous*. To achieve continuity of identity, we identify two properties that key management systems must provide. First, we require *non-equivocation*: we divide time into *epochs* and require that a provider never present more than one binding for a name within an epoch. Second, we require *validity*: the key bound to a name in an epoch must be the same key as in the previous epoch, unless there has been a key change signed by the previous key or the name has been revoked by the user. Taken together, these properties guarantee that there will never be disagreement about which key controls a name at a given time, and that control of a name cannot change unexpectedly.

While some other systems allow equivocation about server certificates to be detected *in principle* (e.g., [33, 48, 61]), no existing identity solutions provide explicit detection of these properties in an *end user-specific* PKI. To this end, we have designed

CONIKS (CONtinuous Identity and Key management System), a system that enables ubiquitous secure communication as well as secure credentials, without users needing to worry about (or even see) cryptographic keys. CONIKS does so by requiring online identity providers to maintain continuous bindings of names to keys. This prevents compelled certificate attacks because any attempt by the provider to associate a new key with an existing name would require a violation of either non-equivocation or validity. CONIKS provides two mechanisms for users to verify these properties, and which guarantee that any violation of either property will be detected promptly and with high probability. Because these mechanisms capture non-repudiable evidence of any violation, users and providers can "whistleblow" on a misbehaving provider by publishing this proof.

In order to achieve scalability, CONIKS requires each participant to perform a relatively modest number of continuity checks on cryptographic commitments made by providers. These distributed checks work together to guarantee that any equivocation or invalidity will be detected promptly with high probability. As a result, any user can be confident that the binding she is seeing for a name is consistent with what another user would see at any point in time.

Through analysis of CONIKS's continuity checks, we find that two users attempting to communicate are capable of detecting equivocation with only *one* honest identity provider in the system. Our evaluation of CONIKS's scalability demonstrates that CONIKS remains highly efficient with millions of providers and billions of users. We have implemented a prototype secure chat service and client which integrate CONIKS for key management.

1.2 Continuity vs. Correctness

In presenting CONIKS, we make the case that *continuity* of a name-to-key binding is both more achievable and more useful than the *correctness* of the binding. First, continuity is easier to achieve. Correctness requires that control of a name corresponds to some real-world notion of accurate naming, which will be difficult to check at reasonable cost in practice. Indeed, individuals have exploited this difficulty to obtain fraudulent certificates for large companies from major CAs [53, 54]. By contrast, continuity can be checked efficiently and automatically. An identity provider can subvert correctness either by error or due to coercion, but failures of continuity will always be detectable, and proof of a continuity failure cannot be repudiated.

Second, at least for the case of identifying end users, continuity is a better match for users' intuitive expectations about naming. Users typically associate an online name with a person and expect that control of the name will not change unexpectedly. Name discovery in CONIKS follows the same conventions that are already in place. With respect to name ownership, there is no *a priori* correct answer to the question of who should control the name *bob@example.com*. What users really expect is continuity. Many of the difficulties of certificate-based identification of users happen because we are trying to use a correctness-motivated design to achieve a continuity property.

Third, because checking and verification of continuity can be automated, users have no need to worry about key management. When a user registers a new account, the user's client software can generate a corresponding key-pair, upload the public key to the identity provider, and wait for the provider to publish a binding connecting the name to the public key. Once this has happened, the user can have confidence that everyone will see her as controlling the name, and that she will continue to control the registered name. The user's control of the name can then be communicated in the same way that names are disseminated now, through a combination of out-of-band

communication and word-of-mouth through third parties. In a continuity-based system, if Alice securely emails Charlie and says that Bob's address is *bob@example.com*, Charlie can record and use this address. This matches Charlie's intuition in the current (insecure) system, in which he treats Alice's statement about Bob's ownership of a name as valid if he trusts Alice. Users can talk about who controls which name—as they do now—and CONIKS will prevent inconsistent views and unexpected transfers of control.

Chapter 2

Related Work

Three recent projects in the areas of certificate validation and identity services are especially similar to CONIKS, and have helped us to further define some aspects of our design for CONIKS. More generally, we draw techniques from four major areas of research: Certificate Validation Systems, Identity Services and Authentication Systems, Alternative Public- Key Infrastructures, and Untrusted Cloud Services,

2.1 Certificate Transparency

A number of projects have proposed systems and protocols for validating SSL/TLS certificates and for making certificate authorities (CAs) accountable for their activity. Certificate Transparency (CT) [32, 33] publicly logs the existence of TLS certificates as they are issued or observed, such that anyone can notice the issuance of suspicious certificates as well as audit the certificate logs themselves. Log services implement the protocol and handle new certificate submissions and queries.

Certificate Transparency's goal is to mitigate the problem of mis-issued certificates by providing append-only untrusted logs of issued certificates. To provide evidence that a certificate has been submitted to a log, the service returns a signed timestamp. CT uses Merkle trees [37] for efficient auditing and to avoid placing trust in the log

servers. Certificates are stored in chronological order in the leaf nodes of the tree. Thus, log servers can provide two kinds of cryptographic proofs. First, they can show that any particular version of the log is an extension of any given previous version. Second, they can show that a particular certificate is present in the tree.

Since Certificate Transparency does not deal with certificate revocation, a few extensions have been proposed to tackle this issue. Revocation Transparency [31] is an informal proposal which describes two mechanisms to prove non-revocation of a certificate in a CT log. These mechanisms used on top of conventional CT, however, would make the protocol much less efficient due to a large amount of additional computation per certificate.

Certificate Issuance and Revocation Transparency (CIRT) [61] seeks to solve the inefficiencies of Revocation Transparency. This CT extension requires log servers to generate two Merkle trees. One serves as the append-only database of certificates as in CT. Revocation of a certificate is signified by adding a null key for the certificate subject. To efficiently prove that a certificate is current, i.e., present in the log and unrevoked, CIRT uses a second Merkle tree which organizes the certificates in lexicographic order. Additionally, CIRT proposes a protocol for transparent key management for end-to-end secure email. In particular, users issue public-key certificates with a CA, subscribe to a certificate prover which maintains a CIRT log, while keeping their existing email address with their email provider; in practice, these three entities could also be represented by a single service provider. In any case, the idea is that users' client software will transparently interact with the CIRT log, avoiding to have to trust any service provider.

CONIKS employs techniques very similar to Certificate Transparency and its extensions, with the added inclusion of explicit handling of equivocation in an efficient manner. CT and CIRT recognize equivocation about certificates by log servers suggesting the use of gossip protocols for clients and auditors to disseminate and compare

Merkle tree root hashes of logs. However, these solutions treat equivocation detection as a separate, external mechanism, and such gossip protocols are thus not part of the CT and CIRT protocols. Furthermore, because we seek to devise a key management system specifically designed for end-users we deem the continuity of identity model to be a more adequate match for users' intuitions about online identity, as we discuss in §1.2. Therefore, although CIRT's and CONIKS's key management protocols have a similar goal, the concepts of identity providers and continuity are more intuitive to users.

2.2 Keybase

A myriad of online identity and authentication services have been developed for various purposes. With the recent increase in demand for end-to-end encrypted communication and authentication, the recently-announced Keybase [29] system is a new way for users to authenticate each other across online services by providing a public directory of publicly-auditable name-to-key bindings.

In Keybase, a user Alice registers a name-to-public key binding, as well as account information from one or more third-party services she controls (e.g., Twitter and Github). Upon registration, Alice digitally signs a tweet and gist confirming the binding between her Keybase identity and her Twitter and Github accounts. When Bob requests Alice's name, whom he knows from Twitter and Github, his Keybase client checks that Alice's listed third-party usernames, belong to the same principal by verifying that the same key was used to generate signatures across her third-party accounts.

Thus, Keybase takes advantage of public identities, and online relationships and reputation, to provide users with secure credentials that can be used across online services. This requires that the two communicating parties in the Keybase model

must both trust common third-party providers, as well as the Keybase client. While this system is very intuitive to users, it does not address the problem of server equivocation. For instance, if Keybase colludes with Twitter, Keybase could generate a bogus key pair for a user, issue the signed tweet, and Twitter could post it on the victim's feed and select whom to show the tweet with the bogus signature.[1]

As we discuss in §3.4, CONIKS can also be used for secure credentials but without requiring users to place any trust in common third-parties. Due to the system's continuity properties, any equivocation about an online identity by the identity providers or the third-party service providers will be quickly detected with high probability. Thus, CONIKS would not be susceptible to attacks by malicious identity providers.

2.3 OneName

Originally designed as a decentralized DNS based on the Bitcoin software, Namecoin [38, 39, 43] supports name registrations, updates and transfers. This makes Namecoin a versatile cryptocurrency that can be used for various other applications including identity systems and messaging systems. In order to register a name, users pay a small amount of Namecoin. The recently-announced OneName [5] service leverages Namecoin to provide a public user directory made of entries in the Namecoin blockchain. Like Keybase, OneName also seeks to provide users with secure credentials to consolidate their online presence at a single account.

In OneName, users register unique global usernames and store their name-to-key bindings in the Namecoin blockchain. Like Bitcoin, Namecoin is a peer-to-peer system and the blockchain grows after each new change in the namespace, which means that the inclusion of a OneName identity in the Namecoin blockchain is distributed to all peers, and verifies that the name registration is authentic. Once in the blockchain, anyone can easily look up a OneName username and obtain the associated public

[1] Such selective presentation of messages has been reported on other social networking sites. [60]

key. Since OneName users can link their name-to-key bindings to accounts with third-party services, users can authenticate each other across online services, much as in Keybase.

Because OneName is a decentralized identity system, users do not have to worry about dealing with certificate authorities or identity providers, which are potentially malicious. On the other hand, the reliance on the peer-to-peer system for confirming name registrations on the blockchain has led to impersonations [49], i.e., two users register the same username but because only one registration is accepted due to the contention to be included in the longer branch of the blockchain, only one user will control this username. Because OneName does not currently provide a mechanism for users to verify that they truly control the name-to-key binding they register, such contention is not discovered until after-the-fact, and potentially malicious users can exploit this conflict.

Impersonation attacks like the ones on OneName may only occur at a local scale in CONIKS since identity providers manage local namespaces and register new name-to-key bindings in batches, but we propose a solution to this issue by the mechanism described in §6.4.2. At the same time, CONIKS is a distributed key management system and leverages this decentralization to detect equivocation about online identities (see §3.3). This enables CONIKS to provide unified online accounts that do not require users to trust their identity providers, as in OneName, but with the added benefits of centralization, which is simpler and requires less of users to join the system.

2.4 Certificate Validation Systems

As mentioned briefly above, one significant research trend has sought to fix some problems with the Certificate Authority (CA) model through certificate validation systems, which aim to reduce the trust in any single CA. Like Certificate Transparency,

Sovereign Keys [47] is another certificate log server. The other main trend in this line of work are certificate observatories such as Perspectives [66],Convergence [58], and SSL Observatory [48].

Certificate observatories create a public repository of SSL/TLS certificates and enable browsers to compare the key they have received with what a small number of trusted observatory servers have witnessed. The Accountable Key Infrastructure (AKI) [27] combines these the techniques used in certificate observatories and log servers to create a certificate validation system which focuses on providing key revocation and accountability.

These approaches all aim to make the certificates of well-known domain names verifiable, and these approaches allow clients to detect equivocation about certificates in principle. CONIKS employs similar techniques as this line of work to provide built-in mechanisms for users to actively and rapidly detect insider attacks, thus focusing on creating a usable PKI for end-user communications.

2.5 Identity Services and Authentication Systems

Before Keybase and OneName, several other projects have sought to provide secure credentials that allow users to consolidate multiple online accounts. The authentication standard OpenID [55] allows a user to create an account with an OpenID identity provider, and then use this account to authenticate themselves to any third-party website that accepts OpenID. This enables single sign-on [65]: a user can access multiple, independent systems without requiring them to log in again at each one.

A myriad of identity services [42, 50, 52, 56] have been built on top of OpenID, the OpenStack Identity API [57], or the OAuth standard [40]. These services offer a method for creating, managing, and authenticating user identities, but they are

typically intended for a specific application or platform [42, 56] or provide tools for developers and businesses to become identity providers [50, 52].

CONIKS is designed in the same spirit as OpenID as it explicitly separates the roles of identity and service providers. However, all of these services remain susceptible to malicious insider attacks.

2.6 Alternative Public-Key Infrastructures

It is well-established that neither the traditional CA model nor the decentralized "Web of Trust" model are perfect solutions to binding certification. Towards this end, PKIs alternative to the traditional X.509 model [25] have been proposed. As a key example, the SPKI/SDSI model [13] emphasizes naming, groups, and flexible authorization. SPKI/SDSI supports disjoint per-provider namespaces and does not give specific semantics to the name bindings, and it exposes the management question of choosing trusted authorities to end-users. Additionally, much like most other PKIs, SPKI/SDSI does not address the security threat of malicious or coerced providers.

SFS [36] introduced the concept of *self-certifying* names: by containing the public key itself (or a hash thereof), a name can separate the issue of key management from the system security. This concept has been widely applied, from network file systems [36] to Internet addressing [8]. Self-certifying names do not require an external key management infrastructure to certify name-to-key bindings, which is ideal in the face of untrusted communication providers. However, self-certifying names are not human-readable, and do not solve the problem of (securely) discovering and disseminating names in the first place.

CONIKS supports disjoint per-provider namespaces and does not give specific semantics to the name bindings like SPKI/SDSI, but because CONIKS is designed to realize the continuity of identity model, it obviates the need for users to make

choices about which authorities to entrust their public keys as these can be held accountable for their misbehavior. Furthermore, while CONIKS cannot avoid relying on an external key management infrastructure, names remain human-readable, and users can securely discover and disseminate secure online identities as these are always verifiable.

2.7 Untrusted Cloud Services

Cloud services are increasingly popular for their ubiquitous accessibility and ease-of-use. However, these cloud services may misbehave or be compelled to equivocate about the content they serve, violating their users' security or privacy expectations. Four recent systems—SUNDR [34], Depot [35], SPORC [16], and Frientegrity [15]–provide explicit mechanisms to detect server equivocation.

These systems either do not explicitly support encrypted content [34, 35], or they assume that an end-user PKI already exists [16, 15]. Furthermore, these systems involve a single, self-contained cloud provider. CONIKS uses similar techniques to make equivocation detectable through tamper-evident data structures, yet applies these techniques to protect the integrity of name-to-key bindings in order to establish a secure, end-user PKI for secure communication. CONIKS also supports a more complex ecosystem that involves federated providers, with potentially interacting users, and where the (mis)behavior of any provider could affect any member of the system.

Chapter 3

CONIKS Overview

Secure email is currently cumbersome since users do not know *a priori* each other's public keys. Instead, standard tools require them to manage and reason about keys, leading to unintentional key leaks and other errors [67]. It is a widely recognized goal to automate key management so that users do not have to worry about it, and cannot do it wrong, but this has not proven possible in conventional PKIs.

In CONIKS users should never have to see encryption keys. CONIKS offers transparent key management enabling secure communication in these simple steps, as shown in Figure 3.1:

1. Alice registers the name *alice* with *foo.com*. In doing so, her client generates a public key that it sends to *foo.com*. *foo.com* publishes a binding that connects the name *alice@foo.com* to Alice's public key, and Alice's client verifies that *foo.com* has done this.
2. When Bob attempts to email *alice@foo.com*, his client looks up the corresponding public key at *foo.com*; this will be Alice's public key.
3. Bob encrypts his message to Alice, signing with his own private key associated with *bob@bar.com*.

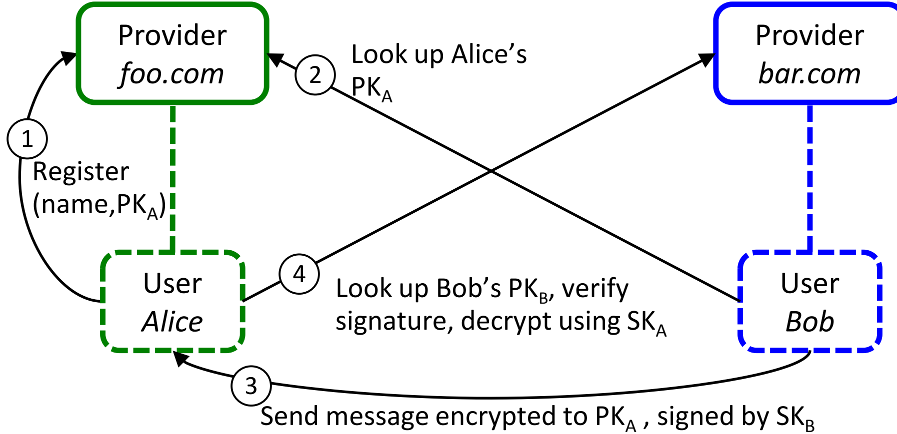

Figure 3.1: High-level interactions between users and identity providers when two users want to communicate securely in CONIKS.

4. Once Alice receives the Bob's message, her client looks up his public key at *bar.com*, verifies the signature on the message, and decrypts the message.

For this to work seamlessly and securely, Alice and Bob must have confidence that the binding that is seen by Bob's client when it looks up the binding for *alice@foo.com* will be consistent with what Alice's client saw when she registered the name. However, this can fail to occur in the two following ways: either *foo.com* can equivocate by giving different bindings to Alice and Bob's clients, or *foo.com* can make an invalid change to the key between the time when Alice's client looks up the name and the later time when Bob's client looks it up.

To guarantee that either type of misbehavior can be detected, CONIKS relies on a set of verifications. Equivocation is detected by having participants randomly query a few providers and compare the answers they receive. Rather than checking individual bindings, this check compares the root hash of a specialized Merkle tree [37] published by each provider. Invalidity is detected by having each user (or an agent acting on the user's behalf) look up its own binding in each epoch. It is sufficient to have a name checked by a single user, because non-equivocation guarantees that any other user who did the same check would see the same result. We detail the checks in §4.2.

3.1 Threat Model

CONIKS's security model reasons about three types of principals: identity providers, users, and third-party entities. We make standard assumptions about the security of cryptographic primitives.

3.1.1 Identity Provider

Identity providers manage disjoint namespaces in which the name-to-key bindings reside. We assume providers use a separate trusted PKI (e.g., X.509) to create non-repudiable statements about the bindings they publish.

An identity provider may be actively malicious or coerced by an outside party to publish arbitrary bindings in response to requests, or to monitor and manipulate the secure communications between specific users. An identity provider cannot defeat cryptographic primitives, but it can learn which users are communicating with each other.

A malicious identity provider might try to violate the confidentiality or integrity of user communications by disseminating a binding that links a user's name to a key controlled by the identity provider. For example, provider *foo.com* could present a client the correct name-to-key binding for the user Alice $(alice, PK_A)$, while presenting $(alice, PK'_A)$ to a user Bob's client, where PK'_A is a public key which *foo.com* controls. In this scenario, any communication from Bob to Alice will use PK'_A, allowing the malicious provider to decrypt confidential messages and forge replies.

A malicious provider also may choose not to respond to client queries. Such an *availability* attack may seek to circumvent detection and verification schemes. We detail the consequences of this behavior in §5.3.

In practice, we expect that CONIKS will deter identity providers from equivocating about their users' bindings since the continuity checks guarantee detection with

high probability (see §5.1). Thus, providers will want to avoid the loss of reputation and business resulting from being caught misbehaving. This consideration of how adversaries will avoid malicious behavior in light of their probability of being detected also appears in the literature introducing the BAR [7] and covert [9] adversary models.

3.1.2 Clients

Users run CONIKS client software, which they may run on multiple trusted devices.[1] To support continuity checks, we assume that at least one of a user's clients has access to a reasonably accurate clock as well as access to secure local storage in which the client can save timestamped information about prior checks. We assume that clients may be unable to communicate directly with each other.

3.1.3 Third-party entities

Entities such as government agencies or third-party adversaries may attempt to gain control over a users' networks or software to mount an active or passive attack on the secure communications between specific users. Recent leaks to the public about government surveillance and collection of user communications data world-wide [11, 20, 24] have revealed that, under the USA PATRIOT Act [6], the FBI can send National Security Letters (NSLs) to communications providers whenever is needs to secretly demand data about American citizens' private communications and Internet activity. NSLs can be sent without oversight or prior judicial review [46, 44, 19], and recipients of NSLs are subject to a gag order which forbids them to reveal the letters' existence.

Thus, the receipt of a NSL is a coercion of a identity provider by the government to equivocate about some or all name-to-key bindings. Since the identity provider is the

[1] We use the terms "user" and "client" interchangeably in this thesis.

entity actually mounting the attack, a user of CONIKS has no way of technologically differentiating between a malicious insider attack mounted by the provider itself and this coerced attack [17]. Nevertheless, because of the continuity checks CONIKS provides, users could expose such attacks as long as there is one identity provider which is not colluding or being compelled (see §5), and thereby mitigate their effect.

3.2 CONIKS Properties

CONIKS enforces the following two properties of name-to-key bindings to provide continuity of identity:

1. **Non-Equivocation**. With respect to a particular epoch, an identity provider must *always* present the identical name-to-key binding to all users and providers.
2. **Validity**. If a name is bound to key PK in some epoch, then in the next epoch either (a) the name is bound to PK again, or (b) the name is bound to another key PK' and a statement authorizing the change and signed by PK is provided, or (c) the name is revoked, and an authorizing statement signed by PK is provided.

To verify these properties, CONIKS provides checks for users and providers to detect violations of continuity. A name-to-key binding is continuous if it passes these continuity checks.

3.2.1 Whistleblowing

If provider misbehavior is detected, CONIKS provides a means for clients to "whistle-blow": to report and publish the evidence of the violation for all identity providers and clients in the system to see. Because providers sign the results they provide, the evidence of the violation will be signed by the provider who is at fault. Not only does whistleblowing alert clients of provider misbehavior, but the absence of a whistle

message is a source of confidence that a provider is being compliant. For instance, if no whistle messages have been reported for Alice's name-to-key binding after a long period of time, Bob can be confident that Alice's identity provider has followed the rules with respect to Alice's name.

3.3 Enabling Continuity Checks

A naive approach to verifying an identity provider's compliance would have each client perform a separate continuity check on each name in the provider's namespace. To avoid the obvious scaling problems with this approach, CONIKS instead has each provider create a Merkle tree containing all of its binding such that non-equivocation checks can be performed over the root of the tree (more details in §4.1). To bind providers to the current state of their entire mapped namespace at a specific point in time, CONIKS requires identity providers to generate *cryptographic commitments* to these *snapshots* at regular time intervals, or *epochs*, by digitally signing the hash of the namespace tree's root node. Due to the properties of Merkle trees, non-equivocation for the root implies non-equivocation for every binding in the tree.

This structure allows us to define CONIKS's continuity properties more precisely:

1. **Non-equivocation about Commitments.** An identity provider must present the identical commitment for an epoch to all users and providers regardless of when they make the request.
2. **Validity of Bindings.** An identity provider must return either the same public key, or information indicating a valid change or revocation, as well as a proof that the name-to-key binding is part of its namespace Merkle tree.

Therefore, clients need only check the validity of those bindings of interest to them, as non-equivocation about the commitments implies that the provider has not equivocated about any bindings. To this end, in a naive approach, clients and

providers would exchange and compare each commitment they see for some specific provider at time t. For example, Alice would send *foo.com*'s commitment in epoch t to Bob, and would waits to receive Bob's version of *foo.com*'s commitment at t. At the same time, Bob would exchange *bar.com*'s commitment with Charlie. This all-pairs communication at each epoch makes the check for non-equivocation quite burdensome; the number of commitment comparisons is on the order of $O(P \cdot N^2)$, where P is the number of providers publishing commitments and N is the total number of clients. To further simplify this check, CONIKS requires that identity providers distribute their commitments to other providers and perform part of the non-equivocation check. Clients then do not need to exchange commitments with other clients, but can merely compare their commitments with what providers are witnessing from others.

In practice, we expect well-behaved identity providers will be willing to perform these checks on behalf of their users as a means to help protect them against other potentially malicious providers. Even today, providers are willing to protect their users' privacy by adopting privacy-enhancing technologies [45], e.g., encrypted SMTP.

3.4 Applications

Our work is motivated by three main applications for CONIKS: Secure webmail, secure online communications, and the secure association of many online identities.

3.4.1 Secure Webmail

CONIKS could be easily integrated into the current user model of webmail services, enabling usable secure webmail backed by continuous email identities. Since each email provider has a unique namespace, webmail providers could act as CONIKS service providers, requiring them to manage the name-to-key mappings for their users,

and to incorporate the CONIKS API and protocols in their services. CONIKS also fits well into notion of email identities that users have: users may have distinct email addresses for different purposes and thus distinct online identities, so CONIKS would allow them to maintain this notion while offering verifiable security. Furthermore, users can notify contacts about their CONIKS identities via the same well-established out-of-band means that are used to announce email addresses, e.g., posting on a blog or handing out business cards.

3.4.2 Secure Communications

While secure webmail is a particularly suitable application of CONIKS, our work is more broadly applicable to any form of online communication which requires users to have a registered name. Online communications span many more forms than just email: instant messaging, social networks, chat rooms, blogs, and even websites. As is the case with secure webmail, prior approaches to secure instant messaging (e.g., Off-the-Record Messaging [10]) and Facebook chat [21] have devised ways of encrypting the messages, but they delegate key management to the users so that they do not have to entrust their data to the communications providers. More importantly, however, is the fact that the notion of continuity of identities remains an unessential or ignored component of such forms of communication online, even in the face of adversaries who are known to equivocate about posts and authorship to surveil or censor users (e.g., [60]). Such communications systems could leverage CONIKS to provide verifiable name-to-key bindings for encrypting or authenticating any messages or published content protecting their users against such attacks.

3.4.3 Secure Credentials

So far we have discussed CONIKS as a system for providing verifiable name-to-key bindings for ensuring webmail and other online communications. However, CONIKS

could also allow users to have secure credentials for various existing online accounts they already own, consolidating them using a single online identity that is compatible across communication providers. For example, the idea is that Alice, who owns both a Twitter account (*@alice*) and a Gmail account(*alice@gmail.com*), could digitally sign all of her tweets and emails she sends from her Gmail address using her CONIKS credentials. At the same time, users could verify that the email they are receiving from *alice@gmail.com* comes from the same online identity which tweets under *@alice*. Thus, CONIKS would provide a means for users to learn which other online accounts are associated with a specific secure online identity, and they could verify the continuity of this identity using the same credentials across services. More importantly, users could use their CONIKS credentials to gain single-sign on [65] access to all of the various accounts associated with a specific online identity.

3.4.4 Adopting CONIKS in practice

We envision two different ways in which developers can integrate CONIKS in the current workflow of communications applications. First, communication providers could themselves become identity providers. In a sense, these providers are already responsible for registering and managing names. Furthermore, users already have the notion of online identities which gain reputation over time. Thus, application developers could integrate CONIKS into their service or create application-specific plug-ins that implement the CONIKS protocols. For instance, developers could create a CONIKS plug-in for a popular email client that transparently allows users to manage name-to-key bindings, performs the necessary validity checks, and incorporates the functionality of existing encryption tools. Second, CONIKS enables creation of "stand-alone" verification services. These services can provide clients that have the same functionality as the application-specific CONIKS plug-ins.

While CONIKS does not require users to interact or manage their contacts' keys, clients must have access to their own private keys. While some implementations of CONIKS clients may expose this directly to users, providers could also support storage of private keys using password-based encryption ([63]). While this makes managing multiple devices easier, it would carry with it the usual trade-offs of password-based encryption schemes. Because of this, CONIKS does not prescribe a specific solution for private key management, and instead leaves this as an implementation choice.

Chapter 4

System Design

Because identity providers may not be truthful about the public keys they report to different users, CONIKS offers two main mechanisms for users and providers to check non-equivocation about commitments and the validity of bindings. When performed every epoch, these checks ensure the integrity of secure communication. At a high level, these mechanisms include the following steps (summarized in Figure 4.1):

1. Alice registers her name *alice* with her identity provider *foo.com* to establish a continuous online identity.
2. At the beginning of the next epoch t, *foo.com* takes a new snapshot of its namespace tree, and generates the signed commitment.
3. Provider *foo.com* publishes this new commitment to all other providers in the system.
4. Upon receiving *foo.com*'s new commitment, provider *bar.com* checks *foo.com*'s commitment history to verify that *foo.com* has not changed the history that *bar.com* has witnessed so far.
5. Alice checks the continuity of her binding in three steps:
   (a) She requests *foo.com*'s newest commitment to check *foo.com*'s commitment history.

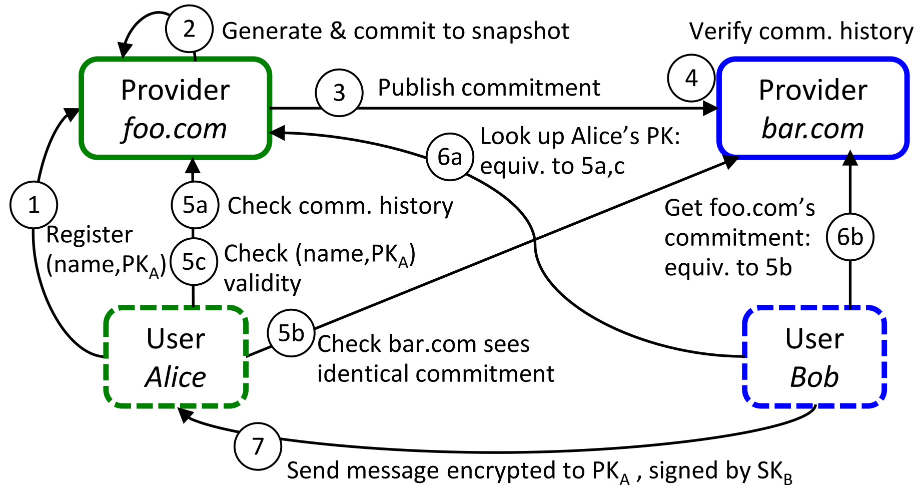

Figure 4.1: Interactions between users and identity providers when performing the continuity checks in CONIKS.

(b) She queries one or more random providers (*bar.com* in this example) for the commitment it witnessed from *foo.com* at the beginning of the epoch. This check is looking for evidence that *foo.com* has equivocated about its new commitment.

(c) She looks up her binding from *foo.com* to check the validity of her binding.

6. Now, Bob, as user of *bar.com*, wants to email *alice@foo.com*. He looks up Alice's public key at *foo.com* and performs the same verifications as Alice in step 5.

7. If Bob detects no misbehavior, he encrypts his message to Alice, and signs it with his private key from *bar.com*. We assume Bob has also verified his own binding in this epoch per step 5.

8. Once Alice receives the message from *bob@bar.com*, she looks up his public key at *bar.com* and checks the key's continuity. If the checks pass, Alice verifies his signature and decrypts the message.[1]

If any of the continuity checks fail, the client whistleblows against the offending provider. Other clients can then query providers for any whistle messages they have witnessed to reduce the number of continuity checks they perform.

[1]We have omitted this last step from Figure 4.1 for simplicity.

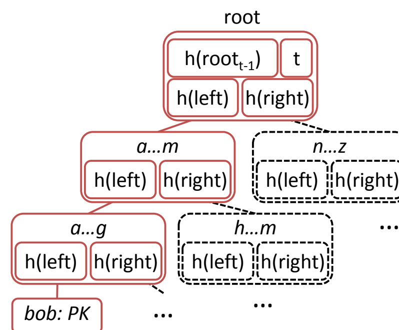

Figure 4.2: An authentication path from Bob's mapping to the root node of the namespace tree. Dotted nodes are not included in the response's authentication path. Internal nodes store the hash of each of their children. Because of the tree's binary-search tree property, users can be confident that only one particular mapping for a key exists in the tree.

4.1 Making Bindings Verifiable

To support the continuity checks that clients and other providers perform, the infrastructure for an identity provider's namespace must satisfy the following requirements:

1. Clients are able to verify the validity of individual bindings.
2. Providers and clients are able to efficiently compare commitments to easily detect equivocation.
3. Commitments are chained so that each epoch's commitment is also committing to a history of all prior commitments from the same provider.

Additionally, identity providers store other providers' commitments which they have witnessed, and maintain a queue of pending updates to the namespace to process these in a batch at the beginning of the next epoch.

4.1.1 Committing to a Merkle Search Tree

Rather than committing separately to each binding, an identity provider constructs an ordered Merkle tree and cryptographically commits to the root hash of the tree. Before verifying that a key is valid, users must be able to tell whether the reported public key is the only key bound to a name. In order to support this, an identity provider's Merkle tree is structured as a binary search tree. When looking up a

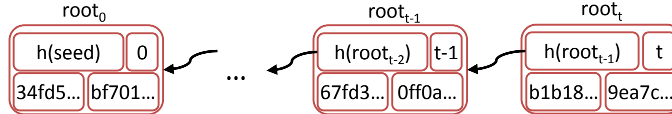

Figure 4.3: `prev` pointers between root nodes form a linear chain of commitments representing the identity provider's history.

particular binding, providers return an *authentication path* (Figure 4.2), i.e., a pruned tree containing the search path to the name-to-key binding. Because of the binary search tree property, users can be sure that only one such path exists.

4.1.2 Comparable Commitments

An identity provider's commitments contain the hash of the provider's tree. This is computed by hashing the value of the root node, which itself contains the hashes of any children nodes. CONIKS leverages this structure to efficiently compare witnessed bindings and commitments to ensure non-equivocation. Since *any* change of a binding results in a change in the value of the commitment, clients performing a validity check can directly compare two commitments and can be sure that the authentication path is consistent with the witnessed commitment.

4.1.3 Associating Commitments over Time

To help detect equivocation more efficiently, identity providers maintain a history of their commitments. Thus, CONIKS clients must have a way of determining whether a provider has presented different users and providers with diverging versions of its commitment history. In order to support this, CONIKS providers include a hash of the *previous* commitment in the current root of their namespace tree, along with the current epoch number. By including the previous hash in the current root node, identity providers create a hash chain of root nodes. (see Figure 4.3).

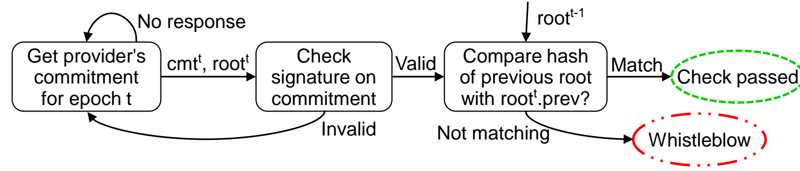

Figure 4.4: Flow chart of the hash chain verification that clients and providers perform every epoch as part of the check for non-equivocation about commitments.

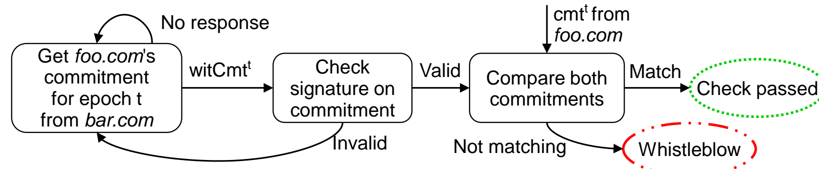

Figure 4.5: Flow chart of the comparison of witnessed commitments that clients perform every epoch as part of the check for non-equivocation about commitments. Identity providers and clients collaborate in verifying that the commitments they are witnessing for a particular identity provider do not differ.

4.2 Continuity Checks

CONIKS provides two mechanisms in which clients and identity providers collaborate to verify, detect, and report any violation.

4.2.1 Checking for Commitment Non-Equivocation

To check that an identity provider is not presenting different commitments to different users and providers, clients and providers verify the hash chain and compare the results of these verifications. Users perform this check whenever their identity provider issues a new commitment. Identity providers, however, perform this check for every commitment which they witness so that users can later compare their version of a commitment with what other providers are seeing to rapidly detect equivocation.

The hash chain verification is summarized in Figure 4.4. To verify the hash chain, the verifier retrieves two commitments: (1) the commitment $(cmt^t, root^t)$ for the current epoch, and (2) the committed root node for the previous epoch $root^{t-1}$. Clients may request these from the identity provider if they are not being cached by the

client. The verifier first ensures that the provider correctly signed cmt^t before checking whether the *previous* pointer in $root^t$ matches the hash of $root^{t-1}$. If these two hashes do not match, the hash chain is invalid and the provider may be attempting to equivocate about its bindings. This invalid hash chain allows the verifier to whistleblow against the provider.

In order to detect whether an identity provider is equivocating, a user will then perform the protocol detailed in Figure 4.5. The user begins by querying a random external identity provider. The client asks this assisting provider *bar.com* for the signed commitment $witCmt^t$ which it has witnessed from provider *foo.com*. The client then compares $witCmt^t$ with the commitment cmt^t which *foo.com* directly presented it. As with other violations, the user whistleblows if this verification fails and the signature is valid.

4.2.2 Checking for Binding Validity

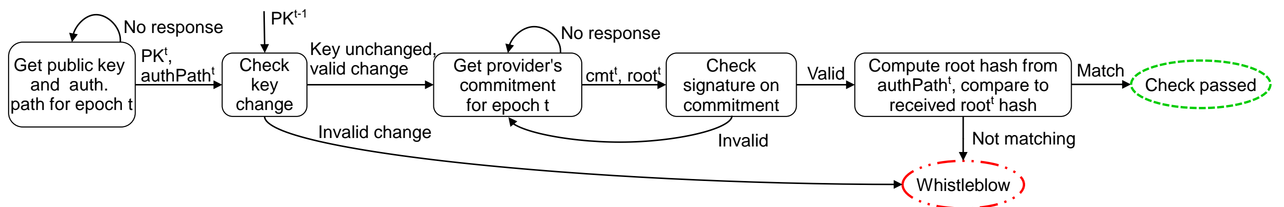

Figure 4.6: Flow chart of the binding validity check a client performs every epoch. Clients first verify whether public key has changed, and then verify the authentication path in the namespace tree.

We outline the check for binding validity in Figure 4.6. To verify that the identity provider returns a valid name-to-key binding, the client begins by looking up the public key for the current epoch PK^t along with the associated authentication path $authPath^t$. The client additionally retrieves the public key for the previous epoch PK^{t-1} from local storage, or requests it from the provider, in order to check that the current key meets one of the three following conditions:

1. The key is unchanged: $PK^t = PK^{t-1}$.

2. An owner-signed and authorized key change occurred.

3. The name has been revoked (owner-signed and authorized).

The client can check the first condition directly. For the second and third conditions, the signed key change, or name revocation, request is included in the server's Merkle tree. A key change request *must* be signed by the old key and must contain the new key; a name revocation request *must* be signed by the user's most recent key. If the binding does not meet either of these conditions, the key change, or revocation, is invalid. A binding is also invalid if a name ever transitions from a *revoked* state to any other state; once a client revokes a name, it is revoked forever. If an invalid key change is detected, the client whistleblows against the provider.

4.2.3 Performing Checks after Missing Epochs

While CONIKS assumes that clients check every epoch, if a CONIKS client is offline or otherwise unable to perform regular checks for some period of time, the client will be able to "catch up" by performing checks which span multiple epochs. Clients who perform these catch-up checks can be just as assured that their providers have not equivocated as if they had been performing regular checks, though equivocations during that offline period may go undetected until the client begins checking again. To perform the catch-up, clients perform the validity check for their binding for each epoch that they missed, and traverse the hash chain through the previous pointers until they have verified that the last witnessed commitment is still included in the provider's most recent commitment. If at any point the hash of the previous root node does not match the previous pointer in the more recently committed root node, the client knows that the provider has changed its commitment history and it whistleblows against the provider.

However, these checks do not, by themselves, imply that the identity provider has not presented other users or providers with different commitments. To verify that the

provider has not equivocated, it is sufficient for clients to compare the current epoch's commitment with what other identity providers witnessed for that same epoch. Because any difference in previous epochs will cause future root hashes to differ (the root's *previous* pointer ensures this), malicious providers must continue to provide diverging views. Therefore, as long as other providers verify the hash chain of each witnessed commitment, clients will be able to detect whether a malicious provider equivocated at *any* point in the past. The client can then use the evidence of the diverging commitments to whistleblow against that provider.

4.3 Whistleblowing

Once a client detects any equivocation during one of the continuity checks, it whistleblows, sending evidence of the continuity violation to another CONIKS identity provider. For example, if Alice, while checking for equivocation, discovers that her provider *foo.com* has published two versions of its commitment history at epoch t, she sends her hash chain computations up to epoch $t-1$ and *foo.com*'s commitment and root node for epoch t to a random identity provider. This demonstrates that *foo.com*'s previous commitment is not included Alice's hash chain computations at epoch t.

Because identity providers and users collaborate in detecting provider misbehavior, identity providers verify and then distribute all whistle messages they receive in an epoch in batches to all other CONIKS providers. Identity providers store any such messages they receive and have distributed to others so that clients can later query them for any evidence of misbehavior by a particular provider that they have recorded.[2] If a user receives evidence of a continuity violation, it can choose to suspend any further continuity checks. Because identity providers distribute whistle messages, the time it takes for a whistle message to propagate through the system is

[2] We have omitted whistleblowing from the continuity checks described in §4.2 for simplicity.

roughly one epoch as identity providers distribute new whistle messages along with their own new commitment. In practice, users may also simply publish the evidence of misbehavior in a public forum where other users can directly retrieve and verify the evidence.

CONIKS's whistleblowing mechanism can deter identity providers from equivocating about name-to-key bindings because evidence of the misbehavior can spread through the system rapidly. In practice, providers will want to avoid the repercussions associated with such misbehavior becoming public, and will therefore be extremely disincentivized from misbehavior.

Chapter 5

Security Analysis

CONIKS's infrastructure and protocols are designed to protect against and detect attacks from identity providers. Thus, CONIKS provides the following guarantees:

- With high probability, users attempting to communicate will rapidly detect any provider equivocation.
- If users can communicate with a *single* trusted provider, any equivocation by their providers will be detected within a single epoch.
- Users will immediately detect any violations of binding validity.

5.1 Detecting Equivocation

If two users, Alice and Bob, wish to communicate, they can be assured that they will obtain the correct public keys for this communication, or that they will detect an attack with high probability within a few epochs. Importantly, Alice and Bob do not need to communicate via an existing secure channel to guarantee this, i.e., even if Bob ends up communicating with a fraudulent Alice, Bob will detect a violation of continuity soon with high probability.

Without violating the validity of name-to-key bindings, malicious identity providers might instead present two different, but independently valid, name-to-key

binding histories, choosing which users and providers will be presented which set of bindings. Even if both histories are valid, the provider has equivocated.

Because CONIKS clients check for equivocation by a provider *foo.com* by contacting other randomly chosen providers and asking them what commitment they have witnessed from *foo.com*, if any client checks with a provider who sees a different commitment than the client saw, then equivocation is detected. However, *foo.com* may be presenting to the providers randomly chosen by Alice and Bob the same history as it presents the two users, respectively, so that neither user detects the attack during this check.

To increase the probability of detecting an equivocation, Alice and Bob can perform multiple rounds of the non-equivocation check comparing their commitments for *foo.com* with those witnessed by multiple randomly chosen providers. Therefore, how many providers must Alice and Bob contact to discover an equivocation about Alice's key with probability α? We will first analyze this for a simple case where Alice's malicious provider *foo.com* is not colluding with other providers, and then we will expand it to the more general case. *foo.com* wishes to present Alice and Bob with different commitments. *foo.com* can choose to present other identity providers either commitment (it could present more commitments, but these only weaken its attack). Suppose that *foo.com* chooses a fraction f of providers who see Alice's commitment, and then $1 - f$ see Bob's. The probability that Alice fails to discover the violation within k checks is f^k, and the probability that Bob fails to discover the violation is $(1 - f)^k$. The probability that both will fail to discover the violation is $(f - f^2)^k$, which is maximized with $f = \frac{1}{2}$. Alice and Bob therefore discover equivocation with probability

$$\alpha = 1 - \left(\frac{1}{4}\right)^k$$

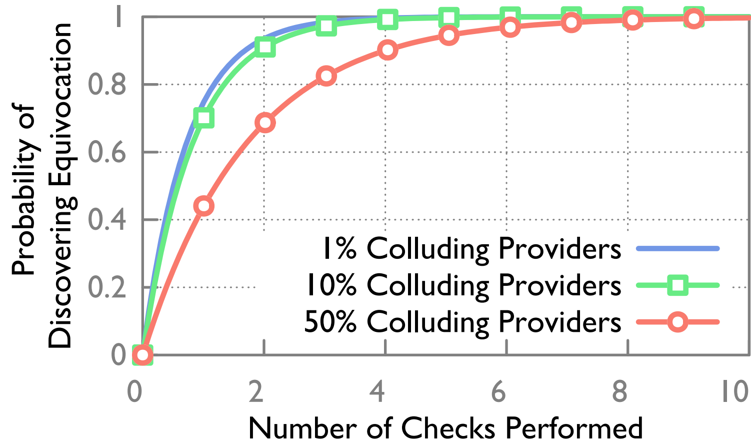

Figure 5.1: Two users, Alice and Bob, will be able to, with high probability, detect any equivocation attacks which attempt to sabotage their communication with a few verification checks. This graph shows the probability that Alice and Bob will detect an equivocation after each performing k checks with random providers.

In order to discover the equivocation with probability α, Alice and Bob must perform $-\log_2(1-\alpha)/2$ checks. After performing 5 checks, Alice and Bob would have discovered an equivocation with 99.9% probability.

In the more general case, if *foo.com* colludes with other providers, allowing the providers to collectively choose which requests see which versions of particular name-to-key bindings, Alice and Bob's verification will become harder to perform. However, as the number of identity providers in the system (and therefore, the number of identity providers not participating in the collusion) increases, the difficulty of verification decreases.

Formally, *foo.com* is colluding with a proportion p of all providers. If Alice queries one of these colluding providers, it will return a commitment which appears valid to Alice. If Bob queries one of these colluding providers, it will return a commitment which appears valid to Bob. Therefore, Bob or Alice must randomly choose a non-colluding provider to discover a continuity violation. The probability of Alice failing to detect equivocation within k checks is therefore $(p+(1-p)f)^k$ and the probability of Bob failing to detect equivocation within k checks is $(p+(1-p)(1-f))^k$. The

probability that neither Alice nor Bob detects equivocation is

$$((p+(1-p)f)(p+(1-p)(1-f)))^k$$

As before, *foo.com* picks $f=\frac{1}{2}$ to maximize this probability. The probability of Alice or Bob discovering this equivocation is

$$\alpha = 1-\left(p+\frac{1-p}{2}\right)^{2k}$$

Figure 5.1 plots the probability of discovery as p and k vary. If fewer than 50% of providers are colluding, Alice and Bob will detect an equivocation within 5 checks with over 94% probability. In practice, large-scale collusion is unexpected, as today's e-mail services have many providers operating with different business models and under many different legal and regulatory regimes.

A single trusted provider. Importantly, if Alice and Bob can communicate with a *single* CONIKS provider that both trust, they will be able to immediately detect any equivocation which occurs between them. If an equivocation between Alice and Bob occurs, this trusted provider will have to have witnessed the same commitment provided to one of Alice or Bob, but not both. Thus, if both communicate with this provider, then an equivocation is guaranteed to be detected. Finding such a trusted provider may not be difficult in practice, as privacy-oriented organizations or Internet watchdogs may operate well-known providers for exactly this purpose.

5.2 Ensuring Validity

All clients will either see all bindings as valid, or immediately detect a violation. An invalid name-to-key binding has either had its key changed without authorization, or two bindings exist in the same commitment for the same name. When a client

requests this tampered-with binding, a provider returns the modified binding, and an authentication path for the binding. If two such bindings exist in the same Merkle search tree, the authentication path would have to be invalid. On the other hand, if the binding has changed without authorization, the misbehaving provider will be unable to respond when the client requests a proof of validity for the key change, because this proof requires an authorizing signature. In both cases, the client will have enough evidence to whistleblow on the provider. As long as the owner of the binding performs the validity check at every epoch, such attacks will be detected.

5.3 Limiting the Effects of Denied Service

Sufficiently powerful identity providers may refuse to distribute commitments to identity providers with which they do not collude. In these cases, clients who query these honest providers will be unable to obtain the two incompatible commitments needed as explicit proof of equivocation. Clients may help circumvent this by submitting witnessed commitments from identity providers to these honest identity providers.[1] An honest identity provider can verify that submitted commitments originated at a particular provider because those commitments contain the original provider's signature.

Similarly, any identity provider may choose to ignore requests about individual bindings with the intentions of preventing the clients from performing validity and uniqueness checks. In these cases, clients may be able to circumvent this attack by using other providers to proxy their requests, with the caveat that a malicious provider may ignore all requests for this name. This renders her identity unusable for as long as the provider denies her service. However, this only allows the provider to deny service for a particular binding, and does not enable the provider to present a new key for that name-to-key binding to other users.

[1] CONIKS does not currently support this feature. We leave this enhancement for future work.

Chapter 6

Discussion

In this chapter, we discuss some of the details of implementing and deploying CONIKS in practice, and some possible protocol extensions.

6.1 Scaling CONIKS

CONIKS's design requires that identity providers perform some additional work when compared to a simple identity service. In particular, identity providers must perform two tasks: communicating commitments at each epoch and verifying other provider's commitments at each epoch. In order to measure the scalability of a straight-forward implementation of a CONIKS identity provider, we performed a set of microbenchmarks. Our results indicate that even with a straight-forward implementation, a single machine is able to quickly handle the additional overhead imposed by CONIKS for workloads similar in scale to the largest identity providers in use today.

6.1.1 Computing and Transmitting Commitments

To understand the computational requirement of computing the hash, we created and signed a Merkle search tree containing 50 million entries, which required 40.24 seconds

on a modern machine. To scale to 1 billion entries, this would require approximately 15 minutes. While 15 minutes may be a significant period of time, this is bearable in practice as identity providers only need to compute this once per epoch and only the largest providers will have over 1 billion entries.

In addition to the computational cost of epoch commitments, identity providers must *transmit* these commitments to each other. Each commitment is composed of the root node and a signature. Using RSA-2048, the size of the signature is 256 bytes. The root node itself contains 4 cryptographic hashes. Using SHA-512, the total size of the commitments is 0.5 KB. For a network of 1 million providers, then, the total upstream and downstream bandwidth requirements are both 488 MB.

6.1.2 Commitment Verification

Before storing newly witnessed commitments from other providers, an identity provider must verify those commitments. This involves checking that the signature matches the supplied root node, and checking that the root node's previous commitment hash matches the previously witnessed root node. Signature verification with RSA-2048 requires 0.16 ms. Computing and comparing the hashes requires 270 ns (not including time to fetch the previously witnessed commitment, which is storage mechanism dependent). In order to verify 1 million new commitments, an identity provider would need to perform 2.67 minutes of computation.

6.1.3 Distributed Operation by Identity Providers.

The semantics of CONIKS require identity providers to always return consistent information. For a well-behaving identity provider running on a single server, this is trivial. The response for a query at any particular epoch should always be the same. However, large identity providers will typically be implemented on a distributed system, such

that updates to bindings are handled by several different (potentially geographically-distant) servers.

One potential implementation is to use a synchronization point between servers at the start of every epoch. Each server would submit the updates it processed since the last epoch to a single master server. This server would then compute the commitment for the next epoch and return it to the other servers. As demonstrated by our simple benchmarks, a single server could perform this task for a identity provider handling over 1 billion bindings. For fault tolerance, this "single" server could also be implemented as a replicated state machine, e.g., running Paxos [30].

6.2 Caching Key Information

While CONIKS does not require that clients store state, today's devices (computers, cellphones, etc.) are capable of caching data. Clients may improve performance by caching the results of prior key lookups. This may result in clients missing account deletion messages, but this is a problem with most revocation schemes. Clients wishing to limit their exposure to revoked keys can regularly flush their cache or choose not to cache keys at all.

6.3 Key Loss and Account Protection

CONIKS users and user clients are responsible for managing their private keys. If users choose not to store private keys locally, they may employ password-authenticated key exchange protocols to store their keys on remote servers or even identity providers. One disadvantage of such mechanisms is that users may forget their passwords or otherwise lose their private keys. CONIKS, however, does not itself provide a mechanism for account recovery since such mechanisms necessarily require out-of-band verification of account ownership, and ultimately require trusting

identity providers to truthfully execute such verifications. A malicious provider could, for example, attack a user Alice by claiming that Alice lost her key and was verified with a new key which is actually controlled by the provider.

However, a user can be sure that even if a key is lost, her identity remains secure; she can continue performing continuity checks on her old binding. Nonetheless, if a future attacker manages to obtain her private key, that attack may be able to assume her "lost identity". In practice, this could be prevented by allowing the provider to issue a name revocation with its own signature, rather than the user's signature. In such cases, the provider would use some specific out-of-band authorization steps to authorize such an action. However, unlike allowing providers to issue key change operations, a name revocation does not require much additional trust in the provider, because a malicious provider could already render an account unusable through denial of service.

6.4 Protocol Extensions

6.4.1 Concealing the Number of Users

Identity providers may not wish to reveal their exact number of users. CONIKS allows providers to insert "null" entries in their binding trees, so long as the entries do not violate the binary search tree property. Such entries can leave the tree unbalanced, or larger than it would have otherwise been. Because of this, providers only expose an upper bound on the number of users they manage. This leads to slightly longer lookups and hash computations. However, this additional overhead would be small.

6.4.2 Returning Updated Bindings between Epochs

Users may desire the ability to use updated bindings *before* the next epoch. Identity providers could support this by maintaining a second "staging" tree, which contains

the updates which have been made during the current epoch, and which guarantees that the bindings are continuous. Users, however, would not be able to perform CONIKS continuity continuity checks for their updated bindings in this staging tree, and would have to wait until the following epoch. When an updated binding is requested, the provider returns the new information, and an authentication path demonstrating that the current epoch's binding is compatible with the new information. The provider signs this message to make it irrefutable.

During the next epoch, the identity provider incorporates the new updates and adds a link from the new epoch's root node to the last staging tree. Any clients which saw an updated binding on the previous day must perform a check to ensure that the witnessed update was correctly incorporated into the epoch. They will check that the binding they received exists in the current epoch and that no conflicting bindings existed in the same staging tree. Any future requests for the binding will be checked as in the normal CONIKS checks.

Chapter 7

Prototype Implementation

To demonstrate the practicality and usability of CONIKS, we have implemented ConiksChat, a prototype XMPP chat service and client which supports Off-the-Record (OTR) Messaging, a secure instant messaging protocol [10]. In particular, we have modified the OTR plug-in [23] for the Pidgin instant messaging client [2] to use CONIKS for key management, and the ConiksChat service leverages the Tigase XMPP/Jabber Server [59] platform for user management.

7.1 Background: XMPP, OTR and Tigase

7.1.1 XMPP

The Extensible Messaging and Presence Protocol (XMPP) [62] provides a generalized, extensible framework for exchanging XML data in near-real-time between two network endpoints. This XML protocol is mainly used to build instant messaging, presence, and request-response services, and is based on Jabber [1], an open instant messaging protocol created in 1998.

Since XMPP is an open standard, not only can any XMPP service implementation may interoperate with other commercial or open-source implementations, but this

protocol allows for inter-operability with other non-XMPP messaging systems through specialized server-side services who translate between XMPP and other messaging protocols (e.g., SMTP, SMS) [62].

7.1.2 Off-the-Record Messaging

This protocol for private social communication is designed to provide perfect forward secrecy and repudiability, two properties the authors of OTR argue are necessary for online social communications [10]. To keep messages private, OTR employs AES encryption, and the encryption key is chosen through a Diffie-Hellman key exchange. Furthermore, to achieve perfect forward secrecy, two users communicating with each other via OTR re-key as frequently as possible to reduce the window of vulnerability when it is possible to decrypt past messages. At the same time, users authenticate each other using digital signatures, and prove authorship of each message using message authentication codes.

The current version of the OTR plug-in (v4.0.0) for Pidgin [23] implements the protocol as follows. Before a user Alice can begin to use OTR, she must generate a DSA key pair through the plug-in, which stores it locally. Then, when Alice wishes to start a private conversation with Bob via OTR, the plug-in performs the Diffie-Hellman key exchange and authenticates Bob in the background. Since Alice's client does not recognize Bob's fingerprint, the plug-in suggests she authenticate Bob.

The plug-in gives Alice two options for authenticating Bob: (1) She can authenticate him via a shared secret [64], or (2) They can manually verify each other's public-key fingerprint. Both Alice and Bob need to perform one of these authentication mechanisms for the conversation to be private. To enhance usability, the plug-in caches all authenticated contacts' public-key fingerprints to avoid the authentication process for later conversations.

7.1.3 Tigase XMPP/Jabber Server

Tigase is an open-source Java implementation of an XMPP/Jabber server [59]. As defined in the XMPP Core RFC [62], this server manages messaging sessions to and from authorized clients as well as connections with other servers via XML streams. This server may also be configured to support storage of user data. More specifically, it can support various relational database management systems for this purpose. The Tigase database schema defines three tables, the most important of which is `tig_users` which contains the main user data such as user ID, password, account creation time.

7.2 ConiksChat Implementation Details

Since XMPP, OTR and Tigase are all open protocols/software, we found them to be the most suitable choices for implementing a first prototype CONIKS identity provider. In particular, we chose to integrate the CONIKS functionality directly with these applications. ConiksChat consists of three main components:

1. The identity provider service (in Java),
2. The client-side CONIKS-OTR plug-in for Pidgin (in C), and
3. Basic CONIKS message protocol buffers [4].

7.2.1 ConiksChat Service

Our ConiksChat service currently supports name registration, public key lookups, and commitment lookups for any epoch. It is important to note that the Pidgin OTR plug-in guided many of our design decisions for this service. As the current implementation of the plug-in employs DSA key pairs for authenticating users, ConiksChat currently only allows for registration of DSA keys. We hope to expand our service to support any kind of public key.

The namespace Merkle search tree is implemented using specially developed tree nodes. The leaf nodes are stored in lexicographic ordering and the tree is padded if the number of nodes is not a power of two. This allowed us to implement the extension of obscuring the number of users that are in the identity provider's namespace. Furthermore, the leaf nodes are stored in a HashMap access to the entire Merkle tree because every tree node has a pointer to its parent node. The provider's snapshot history is stored in a linked list, where each node, or record, contains the following information:

- The epoch of the snapshot,
- The HashMap containing the leaf nodes,
- The root node of the Merkle tree,
- And the signed root hash.

The head of the list is the most recently added snapshot record. This structure facilitates public-key and commitment lookups.

To leverage the Tigase database, and in particular the `tig_users` table, for user registration, we modified the existing schema to include a column for the user's public key as well as an OTR support flag.

Because ConiksChat is currently the only existing CONIKS identity provider, and the number of users in the system is very low, a name registration marks the beginning of a new epoch. To update its history, the identity provider copies the existing name-to-key bindings from the previous version and generates a new Merkle tree including the new binding. It also generates the signature on the root hash at this time.

When a new user registers with ConiksChat, the server first ensures that the user already registered with the Tigase component of the service. If she did, the server checks whether there already exists a name-to-key binding for the received name in its CONIKS namespace. If it does not, the server updates the corresponding database

entry with the received public key and updates its history starting a new epoch, otherwise, the user is notified of the name conflict.

A public key lookup occurs when a ConiksChat client is performing the continuity checks. The server retrieves the snapshot record for the requested epoch, and constructs the authentication path starting at the appropriate leaf node and sends it to the client for verification. Since our server generates the signed root hash of the namespace tree when it updates its history, a commitment lookup merely requires the server to return the root hash and the signed root hash to the client for the requested epoch. However, ConiksChat currently does not distribute the new commitment at the beginning of every epoch, nor does it support witnessed commitments.

7.2.2 ConiksChat Client

The CONIKS-OTR plug-in has required us to make minor modifications to `libotr` [22], the portable OTR Messaging library, as well as make significant changes to the original OTR Pidgin plug-in to support name registration and the continuity checks.

We added the client-side registration mechanism to the DSA key pair generation module of the plug-in, by sending the user's account name and public key to the server after the key pair has been generated. To comply with the OTR key exchange the key pair is also still stored locally. We hope to add support for password-encrypted storage of private keys on the server in the future, avoiding the storage of any keys.

The mechanism for starting a private conversation and exchanging public keys remains unchanged, although we have significantly modified the fingerprint authentication process to perform the continuity checks. Alice's client authenticates her contact Bob's fingerprint in the following steps:

1. The plug-in looks up Bob's public key at the ConiksChat service[1], which returns an authentication path for Bob's name-to-key binding if he has also registered his public key with ConiksChat.
2. The client requests the server's most recent commitment.
3. The plug-in performs the continuity checks on Bob's name-to-key binding using the retrieved information. In particular, the continuity checks, verify the signature on the commitment, check the hash chain, and compute the root hash from the authentication path. If any of the checks failed, Alice is notified.

If Bob has not registered his name-to-key binding with ConiksChat, i.e., he does not use the CONIKS-OTR plug-in for Pidgin, the server will respond to a key lookup with the appropriate error message. The plug-in then performs the original OTR authentication mechanism.

To perform the continuity checks on her own name-to-key binding, a user's client uses the mechanism for contact authentication analogously with her own account name. The continuity checks are triggered when the user first logs into her Pidgin account and the CONIKS-OTR plug-in is enabled. If the checks pass, the client caches the most recent commitment it verified so that the user can still perform the continuity checks, even after a longer period of inactivity. If the any of the continuity checks fail, the client displays a notification warning Alice to proceed with caution, but does not disable messaging. We hope to provide users with a means to revoke their name and whistle-blowing in the case of failed continuity checks on their own bindings.

Similarly as with the server, the client only supports a subset of the CONIKS functionality. CONIKS-OTR is not able to contact randomly chosen providers to check for non-equivocation, nor does it support whistle-blowing.

[1] Because ConiksChat does not currently support key change, any version of the snapshot ought to contain the name-to-key binding Bob first registered. We opt for requesting the information of the current epoch.

7.2.3 CONIKS Protocol Buffers

Lastly, because the client is implemented in C, while the server is implemented in Java, we needed a means to exchange messages between both endpoints. One option was to leverage the existing XMPP infrastructure, but this would have required us to encode all messages into XML in the client and process them in the server. Furthermore, given the nature of the data in CONIKS, i.e., public key material and hashes, a text-based message format would have been inadequate. Thus, we use Google Protocol Buffers (Protobufs) [4] for generating message formats compatible between the client and the server, and which allow us to define specific message types with the required data types for every different kind of message that the client and server exchange.

With the appropriate compiler[2], Protobufs are used to generate message objects (in Java) and message structs (in C) which we then use to serialize and parse messages between clients and the server.

7.2.4 Preliminary Evaluation and Discussion

ConiksChat is currently still in its early stages as it currently does not implement the full CONIKS functionality. However, to see whether our implementation of the service is viable, we ran some preliminary performance tests. In particular, we replicated the microbenchmark discussed in 6.1.1 to measure how long it takes for our server to generate the full Merkle tree. Over 10 trials, it takes around 57.10 seconds on average to compute the Merkle tree with 3 million users, each with a 1024-bit DSA key. This is significantly slower than our simulated microbenchmarks, and the size of the Merkle tree becomes prohibitive for a modern machine for any greater number of users.

While the current implementation is acceptable for a small number of ConiksChat users, this measurement alone tells us that we need to find a better implementation

[2]We use the `protoc-c` compiler [3] to generate our message structs for the client as the Google Protobufs only support compilation into Java, C++ or Python.

for the Merkle tree to make ConiksChat viable in practice. Furthermore, we were only able to successfully implement both continuity checks in simulation, but not in ConiksChat. Currently, the prototype is able to verify non-equivocation properly, but not binding validity. We believe this issue stems from the transition to using Protobufs for representing the authentication path in ConiksChat.

Apart from needing to implement the full CONIKS functionality, our ConiksChat implementation leaves room for improvement in other areas as well. For instance, the server currently does not have any means to restore its state to the most recent epoch in the case of a failure. Additionally, obtaining the public key in the client is significantly slower than in the original OTR plug-in for Pidgin, which does not extract the public key from the generated key pair. Moreover, client-server communication could be done more efficiently. Currently, the client and server tear down the TLS connection after each message exchange. Thus, the TLS handshake incurs additional overhead every time the client performs a continuity check or a lookup.

Chapter 8

Conclusion

Future Work. We would like to develop a more general API for CONIKS to provide a framework for implementing CONIKS identity providers as well as building various kinds of secure messaging applications. There are also a few possible directions for research that we could explore further. We could consider key revocation without account forfeiture and recovery of encrypted data after key loss in the design of CONIKS, two major concerns users have with existing key management systems. Another interesting direction for future work would be to design CONIKS purely as an identity service that provides secure credentials for users across third-party sites.

End-to-end secure communication relies on the ability of parties to discover and use each others' cryptographic keys. Towards this end, today's models for discovering public keys concentrate on establishing "correct" mappings from a key to a named real-world entity and concerns itself with the threat of external attackers. Yet these models do not cope with insider attacks, such as those that coerce a certifier into issuing a bogus certificate for a user.

We introduce the notion of continuity of identity as a basis for preserving secure communication in the face of untrusted providers, and we define two properties that key management systems must provide to achieve such continuity: *non-equivocation*

about commitments and *validity* of name-to-key bindings. Our design for CONIKS, a scalable and usable key management system for end-users, efficiently realizes the continuity of identity model and makes encryption keys invisible to users. CONIKS provides continuity by ensuring that identity providers cannot equivocate in how they answer requests for a named user's keys: either they must always return the same key, or demonstrate that the name has been updated validated to a new key or a revoked state. If an identity provider acts otherwise, CONIKS users and providers detect such continuity violations rapdily, with high probability, and with non-repudiable evidence to serve as a proof of misbehavior.

Our security analysis shows that CONIKS provides this strong security guarantee while requiring clients to store little to no state. Furthermore, only a single honest provider is necessary to detect misbehavior, although the probability of detection improves exponentially with the fraction of honest providers. Our prototype chat service and microbenchmarks demonstrate that CONIKS is practical: it integrates naturally into the existing Internet and Web ecosystem, and it scales to millions of providers and billions of users.

Bibliography

[1] Jabber.org. http://www.jabber.org, Retrieved Apr. 2014.

[2] Pidgin. http://pidgin.im, Retrieved Apr. 2014.

[3] protobuf-c. https://github.com/protobuf-c/protobuf-c, Retrieved Apr. 2014.

[4] Protocol Buffers. https://code.google.com/p/protobuf, Retrieved Apr. 2014.

[5] OneName. https://onename.io, Retrieved May 2014.

[6] USA PATRIOT Act of 2001. http://www.gpo.gov/fdsys/pkg/PLAW-107publ56/pdf/PLAW-107publ56.pdf, Retrieved Nov. 2013.

[7] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proc. SOSP*, Oct. 2005.

[8] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable internet protocol (aip). In *Proc. SIGCOMM*, Aug. 2008.

[9] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptol.*, 23(2):281–343, April 2010.

[10] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proc. WPES*, Oct. 2004.

[11] Stephen Braun, Anne Flaherty, Jack Gillum, and Matt Apuzzo. Secret to Prism program: Even bigger data seizure. http://bigstory.ap.org/article/secret-prism-success-even-bigger-data-seizure, Jun. 2013.

[12] J Callas, L Donnerhacke, H Finney, and R Thayer. RFC 2440 OpenPGP Message Format, Nov. 1998.

[13] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, Dec. 2001.

[14] Paul Everton. Google's Gmail Hacked This Weekend? Tips To Beef Up Your Security. http://www.huffingtonpost.com/paul-everton/googles-gmail-hacked-this_b_3641842.html, Jul. 2013.

[15] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proc. USENIX Security*, Aug. 2012.

[16] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: group collaboration using untrusted cloud resources. In *Proc. OSDI*, Oct. 2010.

[17] Ed Felten. A Court Order is an Insider Attack. https://freedom-to-tinker.com/blog/felten/a-court-order-is-an-insider-attack/, Oct. 2013.

[18] Shirley Gaw, Edward W. Felten, and Patricia Fernandez-Kelly. Secrecy, flagging, and paranoia: Adoption criteria in encrypted email. In *Proc. CHI*, Apr 2006.

[19] Barton Gellman. The FBI's Secret Scrutiny. http://washingtonpost.com/wp-dyn/content/article/2005/11/05/AR2005110505366.html, Nov. 2005.

[20] Barton Gellman and Laura Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html, Jun. 2013.

[21] Stefan George, Felix Leupold, and Carolyn Estrada. BlockPRISM. http://www.indiegogo.com/projects/blockprism-org/, Retrieved Aug. 2013.

[22] Ian Goldberg, Katrina Hanna, and Nikita Borisov. libotr. http://sourceforge.net/p/otr/libotr/ci/master/tree/, Retrieved Apr. 2014.

[23] Ian Goldberg, Katrina Hanna, and Nikita Borisov. pidgin-otr. http://sourceforge.net/p/otr/pidgin-otr/ci/master/tree/, Retrieved Apr. 2014.

[24] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google and others. http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data, Jun. 2013.

[25] R. Hously, W. Ford, W Polk, and D. Solo. RFC 2459 Internet X.509 Public Key Infrastructure, Jan. 1999.

[26] Alastair Jamieson and Erin McClam. Millions of Target customer's credit, debit card accounts may be hit by data breach. http://www.nbcnews.com/business/consumer/millions-target-customers-credit-debit-card-accounts-may-be-hit-f2D11775203, Dec. 2013.

[27] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perring, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *Proc. WWW*, 2013.

[28] Gabrielle Kratsas and Karl Gelles. Target breach helps usher in new world of data security. http://www.usatoday.com/story/money/business/2014/02/22/retail-hacks-security-standards/5257919/, Feb. 2014.

[29] Max Krohn and Chris Coyne. Keybase. https://keybase.io, Retrieved Feb. 2014.

[30] Leslie Lamport. The part-time parliament. *Trans. Computer Systems*, 16(2), 1998.

[31] Ben Laurie and Emilia Kasper. Revocation Transparency. http://sump2.links.org/files/RevocationTransparency.pdf, Retrieved Feb. 2014.

[32] Ben Laurie, Adam Langley, and Emilia Kasper. RFC 6962 Certificate Transparency, Jun. 2013.

[33] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. http://www.certificate-transparency.org, Retrieved Aug. 2013.

[34] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.

[35] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michae Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.

[36] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proc. SOSP*, Dec. 1999.

[37] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proc. CRYPTO*, Aug. 1987.

[38] namecoin1. Namecoin. http://namecoin.info/, Retrieved Nov. 2013.

[39] namecoin2. Namecoin. https://en.bitcoin.it/wiki/Namecoin, Retrieved Nov. 2013.

[40] OAuth. http://oauth.net/, Retrieved Nov. 2013.

[41] Nicole Perloth. Yahoo Breach Extends Beyond Yahoo to Gmail, Hotmail, AOL Users. http://bits.blogs.nytimes.com/2012/07/12/yahoo-breach-extends-beyond-yahoo-to-gmail-hotmail-aol-users/, Jul. 2012.

[42] Hewlett-Packard Development Company, L.P. HP Cloud Identity Service. http://www.hpcloud.com/products-services/identity-service, Retrieved Nov. 2013.

[43] DotBIT Project. Namecoin. https://github.com/namecoin/namecoin, Retrieved Nov. 2013.

[44] Electronic Frontier Foundation. National Security Letters - EFF Surveillance Self-Defense Project. https://ssd.eff.org/foreign/nsl, Retrieved Aug. 2013.

[45] Electronic Frontier Foundation. Encrypt the Web Report: Who's Doing What. https://www.eff.org/deeplinks/2013/11/encrypt-web-report-whos-doing-what, Retrieved Feb. 2014.

[46] Electronic Frontier Foundation. National Security Letters. https://www.eff.org/issues/national-security-letters, Retrieved Nov. 2013.

[47] Electronic Frontier Foundation. Sovereign Keys. https://www.eff.org/sovereign-keys, Retrieved Nov. 2013.

[48] Electronic Frontier Foundation. SSL Observatory. https://www.eff.org/observatory, Retrieved Nov. 2013.

[49] hightorque. OneName:the Decentralized Identity System for Bitcoin. http://www.reddit.com/r/Bitcoin/comments/201g66/onename_the_decentralized_identity_system_for/, Retrieved May 2014.

[50] Intel Corporation. Intel Identity Services. http://software.intel.com/cloudservicesplatform/service/intel-identity-services, Retrieved Nov. 2013.

[51] Internet Mail Consortium. S/MIME and OpenPGP. http://www.imc.org/smime-pgpmime.html, Retrieved Aug. 2013.

[52] Janrain, Inc. Janrain. http://janrain.com/products/identity-service/, Retrieved Nov. 2013.

[53] Microsoft Corportation. How to Recognize Erroneously Issued VeriSign Code-Signing Certificates. http://support.microsoft.com/kb/293817, Feb. 2007.

[54] Microsoft Corportation. MS01-017: Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard. http://support.microsoft.com/kb/293818/en-us, Feb. 2007.

[55] OpenID Foundation. OpenID. http://openid.net/, Retrieved Nov. 2013.

[56] OpenStack, LLC. Keystone. http://docs.openstack.org/developer/keystone/, Retrieved Nov. 2013.

[57] OpenStack, LLC. OpenStack Identity Service API v2.0 Reference. http://docs.openstack.org/api/openstack-identity-service/2.0/content/, Retrieved Nov. 2013.

[58] Thoughtcrime Labs Production. Convergence. http://convergence.io, Retrieved Aug. 2013.

[59] Tigase, Inc. Tigase XMPP/Jabber Server. http://www.tigase.org, Retrieved Apr. 2014.

[60] Reuters. At Sina Weibo's Censorship Hub, 'Little Brothers' Cleanse Online Chatter. http://www.voanews.com/content/reu-sina-weibo-censorship-online-chatter/1748103.html, Retrieved Nov. 2013.

[61] Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted email. In *Proc. NDSS*, Feb. 2014.

[62] P. Saint-Andre Ed. RFC 3920 Extensible Messaging and Presence Protocol (XMPP): Core, Oct. 2004.

[63] S. Shin and K. Kobara. RFC 6628 Efficient Augmented Password-Only Authentication and Key Exchange for IKEv2, Jun. 2012.

[64] Ryan Stedman, Kayon Yoshida, and Ian Goldberg. A user study of off-the-record messaging. In *Proc. SOUPS*, Jul. 2008.

[65] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proc. SP*, May 2012.

[66] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: improving SSH-style host authentication with multi-path probing. In *Proc. ATC*, Jun. 2008.

[67] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *Proc. USENIX Security*, Aug. 1999.

[68] Philip R. Zimmermann. *The official PGP user's guide.* MIT Press, Cambridge, MA, USA, 1995.